# A DYNAMIC PERFECT HASH FUNCTION DEFINED BY AN EXTENDED HASH INDICATOR TABLE

W. P. Yang and M. W. Du

Institute of Computer Engineering
National Chiao Tung University
Hsinchu, Taiwan, ROC

ABSTRACT — This paper presents a new dynamic file organization scheme based on hashing. The hash functions used here, being defined by extended hash indicator tables (EHITs), are both dynamic and perfect. The allocated storage space can be enlarged and shrunk without reorganizing the data file. Simulation results show that the storage utilization is approximately equal to 70% in an experiment where the number of rehash functions s=7, the size of a segment r=10, and the size of the key set n varies from 1 to 1000. Since the hash functions are perfect, the retrieval operation needs only one disk access.

## 1. INTRODUCTION

Hashing is a fast technique for information storage and retrieval [12,19,20,22,25]. Its use, however, may cause some problems. First of all, the user needs to handle key collision problem. Secondly, the address space cannot easily be changed dynamically. The former problem may be solved by the use of perfect hash functions, such as those proposed in [1,2,3,5,6,7,9,10,26,28,30], where a perfect hash function is defined as a one-to-one mapping from the key set into the address space. The latter problem may lead to a waste of memory if the address space is too large or to a poor performance if the address space is too small [12]. Thus, dynamic allocation of the address space is needed. Dynamic hashing means that in the hashing scheme the set of keys can be varied; i.e., keys can be inserted into or deleted from the key set. Insertion of a key may lead to split operations -- each split allocates one segment to the address space -- while deletion of a key may lead to deallocation of segments which reduces the address space in use.

Several dynamic hashing schemes have been introduced during the last few years: expandable hashing by Knott [11], dynamic hashing by Larson [13], virtual hashing, linear virtual hashing, and trie hashing by Litwin [16,17,18], and extendible hashing by Fagin and others [8]. The detailed operations of these schemes are different, many of them use a similar dynamic auxiliary table to define or modify the hash functions. The auxiliary table is referred to as "index tree" [13], "bit-map table" [16], "prefix tree" [4,11], or "directory" [8]. These schemes can be divided into two classes. One uses overflow area in address space and the other does not. Generally, those schemes using overflow area, save the memory in the address space but need one or more disk access to retrieve a key. Virtual hashing [16] and dynamic hashing with deferred splitting [24] belong to this class, and the expected storage utilization of them is about 80%, under suitable conditions. The second class are those not using overflow area which can retrieve a key with just one disk access. However, storage utilization is lower, say only about 65-69% under similar conditions. Dynamic hashing [13] and extendible hashing [8] belong to this class.

In this paper, we design a new dynamic hash function belonging to the second class. The storage utilization exceeds 69% and the auxiliary table, which modifies the hash functions, is relative smaller. The main idea is to apply the hash indicator table (HIT) to define perfect hash functions as proposed in [6,7]. A modified HIT called extended hash indicator table (EHIT) is employed as an auxiliary table to set the dynamic hash functions. Following is a brief description of how to use HIT to define a perfect hash function.

### Define Perfect Hash Function by HIT

In [6,7] a perfect hash scheme that solves the key collision problem by using a series of hash functions is proposed. Because the HIT stores the index of hash functions selected, it can define a perfect hash function. The basic model of this scheme is shown in Figure 1.1. The perfect hash function h can be defined by HIT as follows [6,7].

$$h(k_i)=h_j(k_i)=x_i \text{ if } HIT[h_r(k_i)]\neq r \text{ for } r < j \text{ and } HIT[h_j(k_i)]=j,$$
$$=\text{undefined otherwise.}$$

The advantages of the perfect hash functions defined by HIT here are that they can be easily

*Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.*

**Proceedings of the Tenth International Conference on Very Large Data Bases.** **Singapore, August, 1984**

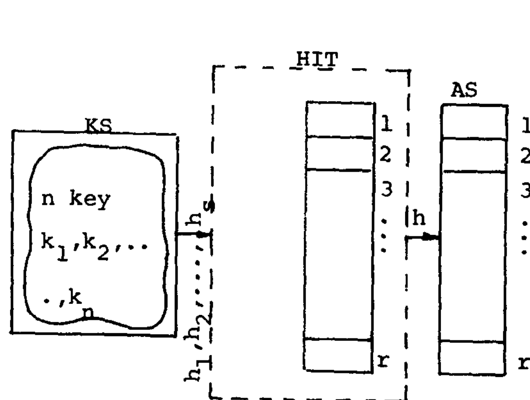


Fig. 1.1 The basic model of perfect hash functions defined by HIT.

implemented and they use very small tables. Three ways to construct HIT have been proposed. They are process-by-function method [6], process-by-key method [28], and branch-and-bound method [30]. We shall discuss these methods briefly in Appendix. The retrieval algorithm is very simple. We list it below:

```
Procedure RETRIEVAL(k,s,HIT,AS);
//Assume that the hash functions h1,h2,//
//...,hs, are used to create HIT. Now //
//we want to retrieve key k.//
begin
  j:=1;
  while (HIT[hj(k)]≠j and j≤s) do j:=j+1;
  if j > s then failure
         else k is stored in AS[hj(k)]
end.
```

In evaluating the function value, the number of loops executed in the while-statement in the algorithm is called the "number of internal probes" in HIT. The expected number of internal probes is defined by $1/n \sum_{i=0}^{r-1} HIT[i]$, where n is the number of keys concerned and r is the size of the address space. It is a measure of the retrieval cost.

In Section 2 we will describe how to use a modified scheme to handle dynamic file. Insertion and deletion algorithms will be presented. In Section 3, the expected performances will be considered. We will use independent data to simulate the scheme and to observe the variation of address space. Also we will estimate the utilization of storage and the retrieval costs.

## 2. RETRIEVAL AND MAINTENANCE ALGORITHMS

### 2.1 Define Dynamic Perfect Hash Function by EHIT

The model of dynamic perfect hash functions in this new file retrieval system is shown in Figure 2.1. It consists of an address space (AS) in which the keys are stored, and an extended hash indicator table (EHIT) as auxiliary memory. The EHIT contains two parts: the distribution tree(DT) and HIT.

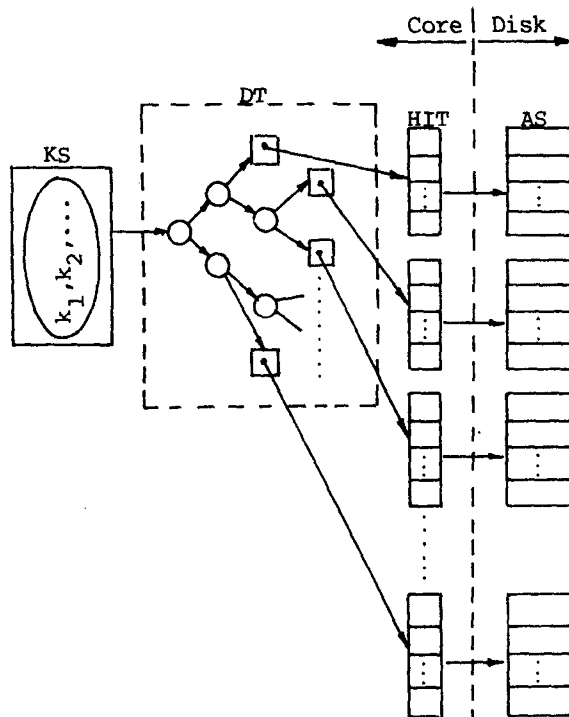


Fig. 2.1 Model of a dynamic perfect hash file.

Assume that the set of keys to be stored at a certain time is denoted by $\{k_i\}$, i=1,2,...,n. The number of keys, n, is not fixed but varies with time. The system is initialized with a node and a segment of HIT with predefined size, r. Secondary storage space is allocated for one segment with capacity r. Each segment of HIT contains a pointer to the segment of AS in the data file. Figure 2.2 illustrates the initial situation for a file with one segment.

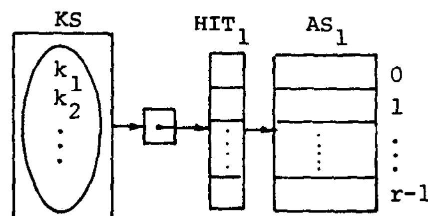


Fig. 2.2 Initial structure of a dynamic perfect hash file.

First some keys are placed into the system and HIT is constructed by using the branch-and-bound method [30] (see Appendix) to define the perfect hash function.

Sooner or later the segment will overflow, i.e., when trying to insert a key into the segment, the corresponding HIT cannot be constructed successfully. When this happens we split the segment into two. A new segment of address space is allocated and the keys are distributed equally among the two segments. At the same time the DT is updated to record the new situation. If, later, one or the other of the two segments becomes "full", it splits into two segments, etc. Figure 2.3 shows the structure of the hash file derived

from Figure 2.2 after three splits have occurred. The size of the file has been increased to four segments. The DT has grown to a binary tree with seven nodes. Internal nodes are shown as circles and external nodes as squares. Under this scheme, the retrieval operation is as follows:

```
Algorithm DYNAMIC_RETRIEVE(k,s,DT,HIT)
begin
  //traverse DT//
  //call random number generator//
  b1b2...bm:=RANDOM(k:seed)
  i:=1
  while not external node do
     if bi=0
     then goto left-branch else goto right-
     branch;
     i:=i+1
  //get HITj and find hi//
  i:=1;
  while HITj[hi(k)]≠i do i:=i+1;
  if i > s
  then failure else k is stored in
  ASj[hi(k)];
end.
```

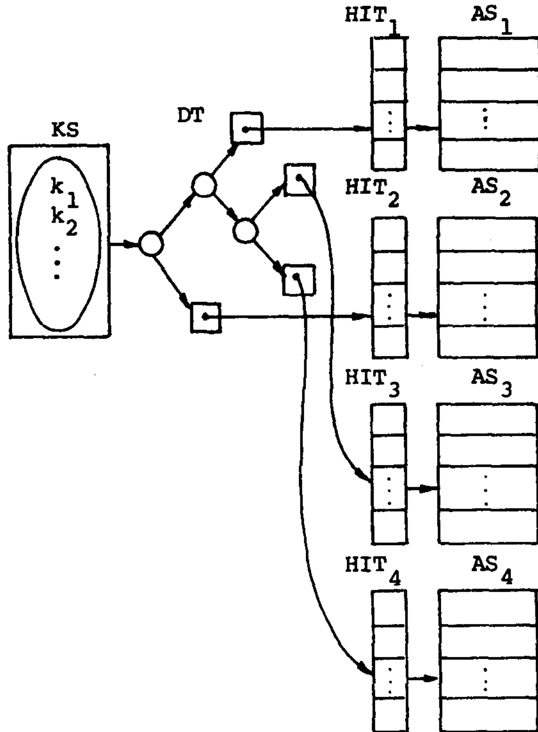


Fig. 2.3 Structure of a dynamic perfect hash file after three splits.

There are mainly two advantages. As one can see, the algorithm is very simple. Secondly if we can put the EHIT into the core memory, we can retrieve a key with only one disk access.

## 2.2 Insertion and Growth

### Splitting

Here is an example to illustrate the behavior of the splitting in this dynamic hash scheme.

Example 1. Let the key set KS,

KS={AND,END,NIL,SET,ARRAY,BEGIN,CASE,CONST, ...},

be the 36 Pascal's reserved words. The values of three hash function and two random numbers are listed in the following table:

| | AND | END | NIL | SET | ARRAY | BEGIN | CASE | CONST | ... |
|---|---|---|---|---|---|---|---|---|---|
| $h_1$ | 3 | 1 | 5 | 0 | 1 | 1 | 4 | 0 | ... |
| $h_2$ | 4 | 4 | 3 | 3 | 0 | 1 | 3 | 1 | ... |
| $h_3$ | 1 | 1 | 0 | 5 | 3 | 4 | 4 | 5 | ... |
| random value $bit_1$ | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | ... |
| random value $bit_2$ | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | ... |

The hash functions $h_i$'s are calculated by

$$h_i(key)=[ORD(ch_i)+ORD(ch_{i+1})*i] \bmod 6$$

where $ORD(ch_i)$ denotes the ordinal number of the ith character of key. When $ch_i$ is the last character of the key word, $ch_{i+1}$ resets to the first character of the key. The random values are obtained by calling a random number generator, which uses the decimal codes of each key as seed.

(1) We will process on the keys in the order of $k_1$,$k_2$, etc. First, we process $k_1$ and show the configuration as in Figure 2.4(a).

(2) After processing the first four keys, the current size of key set is four. We get Figure 2.4(b) by using the same branch-and-bound method to construct HIT as proposed in [30] (see Appendix).

(3) When we want to insert the 5th key ARRAY, we cannot define a perfect hashing function in only one segment $HIT_1$. Then the second segment $HIT_2$ is allocated. The keys are distributed into these two segments according to whether the bit's value is 0 or 1. The confi-

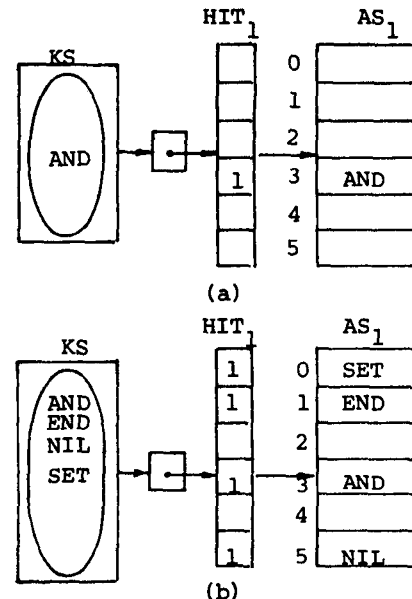


Fig. 2.4 Configuration of the dynamic hash file.

guration is depicted in Figure 2.4(c).

(4) After processing the eight keys, the distribution tree has been split twice, and three segments of HIT and AS are allocated. The configuration is shown in Figure 2.4(d). Notice that the second split causes the $HIT_1$ to split into $HIT_1$ and $HIT_3$, but $HIT_2$ and $AS_2$ are not changed at all.

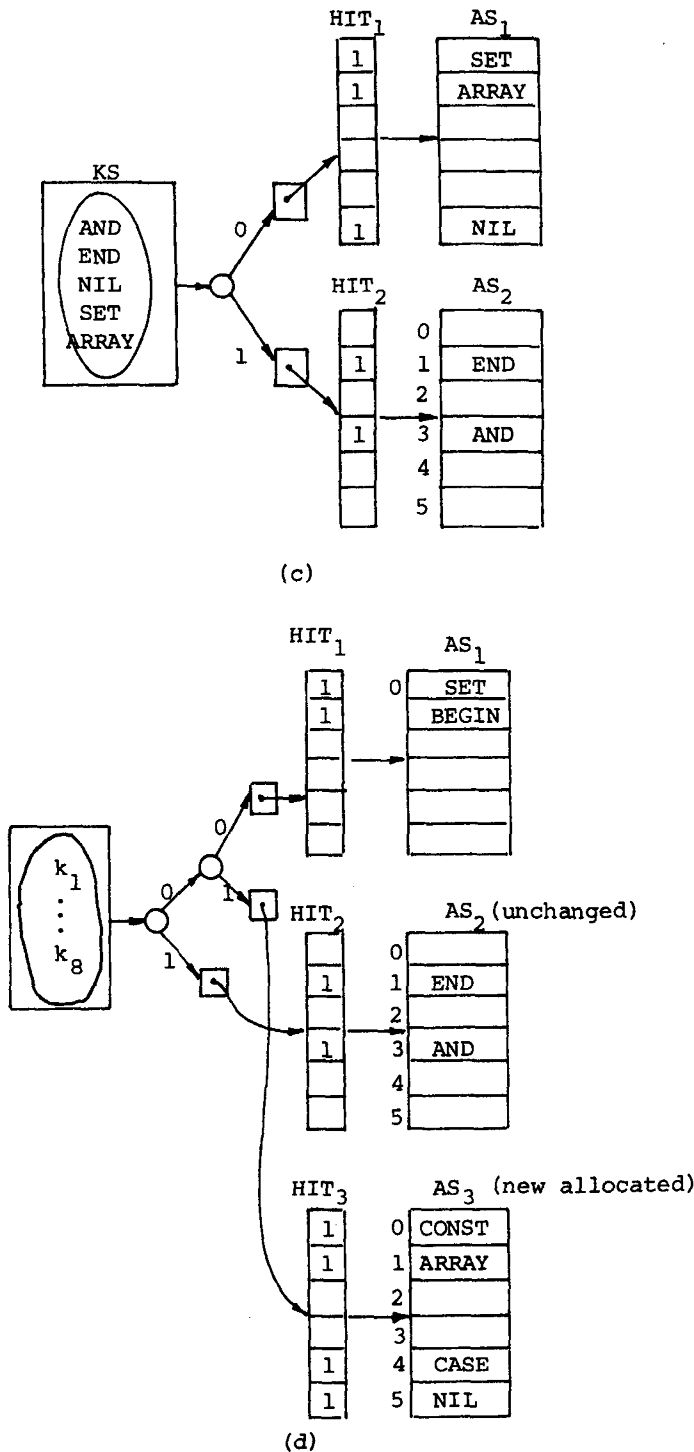


Fig. 2.4 Configuration of the dynamic hash file. (continued)

The general algorithm for splitting is shown, with Figure 2.5 below.

```
Algorithm SPLIT(ptr)
begin
   1. get HIT_i, AS_i pointed by ptr
   2. TEMP:=AS_i
   3. get a new node HIT_j, a new node for AS_j
   4. distributes TEMP into AS_i and AS_j
   5. Using branch-and-bound method to
      construct HIT_i with AS_i and HIT_j with
      AS_j
   6. update DT
end.
```

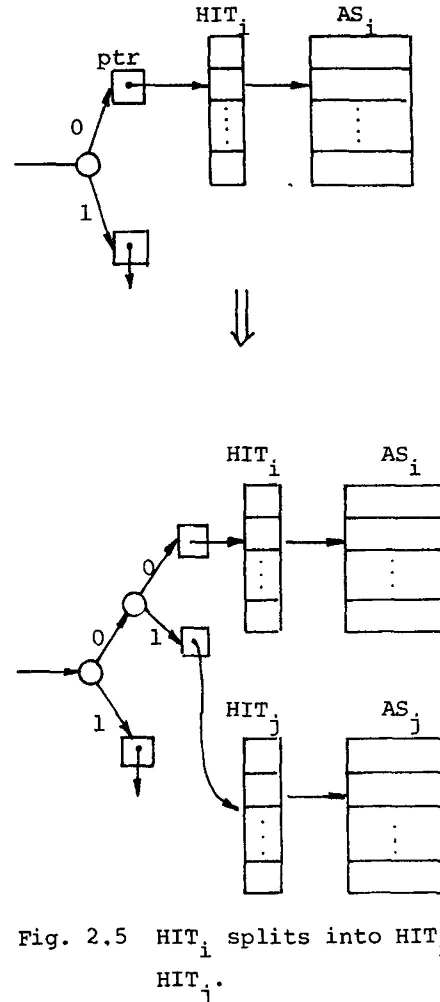


Fig. 2.5 $HIT_i$ splits into $HIT_i$ and $HIT_j$.

Here we discuss some properties of splitting. Let X be a random variable to represent the number of keys that are put together into the same segment after a split. We have

$$\text{Prob}(X=x)=2\binom{n}{x}(0.5)^x(1-0.5)^{n-x}$$
$$=\binom{n}{x}0.5^{n-1}.$$

If n=x,

$$\text{Prob}(X=n)=0.5^{n-1}$$

is the probability that all the n keys will be put into the same segment. For example, n=5, Prob(x=5)=0.0625. This implies, the probability of all the keys being put into the same segment

is very small.

Let Y be a random variable such that all the n keys are put into the same segment after t splits. Then,

$$Prob(Y=t)=0.5^{(n-1)(t-1)} \cdot (1-0.5^{n-1}).$$

For example, if n=5, Prob(Y=3)=3.66 x $10^{-3}$, is very small.

### Insertion

The general insertion algorithm that uses the SPLIT procedure described above is listed as follows:

```
Algorithm INSERT(k,DT,HIT)
begin
  1. b1b2...bm:=RANDOM(k:seed)
  2. traverse the DT depending on bi
     until external node E
  3. get the HITi pointed by E
  4. get the ASi pointed by HITi
  5. using the branch-and-bound method re-
     construct HITi
  6. if step 5 done then return
                    else call SPLIT
end.
```

### 2.3 Deletion and Shrinkage

Deletion of a key may cause a segment of address space to become empty. The hash file can be shrunk by

(1) deallocating the empty segment,
(2) freeing the corresponding HIT to the free storage pool, and
(3) updating the DT, which eliminates two nodes in general.

In Figure 2.6, when $K_4$ and $K_6$ are deleted, $AS_1$

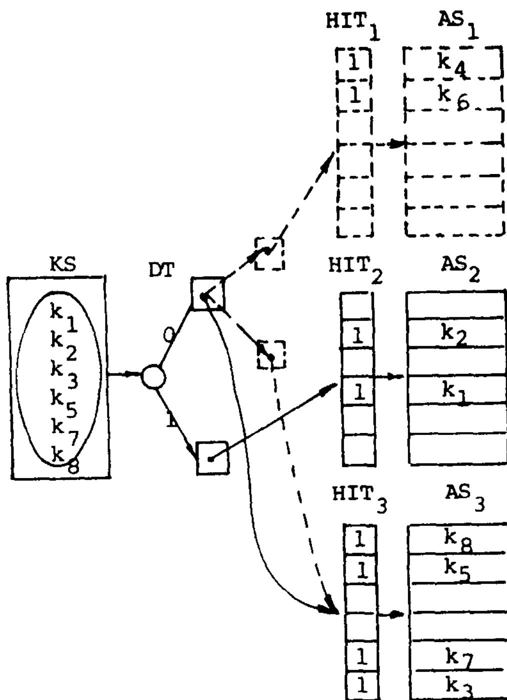


Fig. 2.6. After deleting $k_4$ and $k_6$, $HIT_1$ and $AS_1$ are deallocated, while the DT is updated with two nodes eliminated.

becomes empty, then $AS_1$, $HIT_1$ and two nodes of DT are deallocated. This means the hash file is shrinking. The general algorithm of deletion is:

```
Algorithm DELETION(k,DT)
begin
  1. traverse DT to find the HITi
     such that k ∈ ASi
  2. delete k from ASi, reset HITi
  3. if ASi = empty
     then 3.1 deallocate ASi
          3.2 deallocate HITi
          3.3 update DT
end.
```

## 3. EXPECTED PERFORMANCES

### 3.1 Analysis

In this section we mainly consider the height of the DT and the storage utilization of a dynamic hash file. Assuming that the file is empty, then it is created by n insertions, and the size of each segment of AS and HIT is r. We will also assume that each $h_i$ hashes randomly and that the DT is represented by a binary tree using linked structure. We shall start by analyzing the number of nodes in the DT. Then, we can obtain the number of allocated segments, the storage utilization, and the height of the DT immediately. Here we adapt the analysis given in [13] to our scheme.

Dt is a binary tree. The number of nodes on level t is $2^t$, $t \geq 0$. The probability of the search sequence of certain record with key k passing through a given node on level t is

$$1/2^t, \; t \geq 0.$$

The probability that x out of n keys pass through the node is then

$$P_t(x)=\binom{n}{x}(1/2^t)^x(1-1/2^t)^{n-x}, \; 0 \leq x \leq n.$$

The x keys pass a node if the y keys pass is father node first. The conditional probability is:

$$\binom{y}{x}(1/2)^x(1-1/2)^{y-x}$$

$$=\binom{y}{x}(1/2)^y, \; y \geq x.$$

From the above probabilities, we obtain the probability that a node at level t such that y keys pass its father and then x key pass this node is:

$$Q_t(x)=\sum_{y=r'+1}^{n} P_{t-1}(y)\binom{y}{x}1/2^y, \; 0 \leq x \leq n, \; t>0.$$

$$Q_0(x)=P_0(x), \qquad t=0.$$

where r' is the maximum number of keys in one segment. If $0 \leq x \leq r'$, then a node at level t does not split. The probability is

$$R_t = \sum_{x=0}^{r'} Q_t(x), \; t \geq 0.$$

The probability that a node at level t points to a segment of HIT is

$$R'_t = \sum_{x=1}^{r'} Q_t(x), \; t \geq 0.$$

The expected number of nodes at level t which point to a segment of HIT is

$$Exp(R'_t) = (\text{the number of nodes at level } t) \cdot R'_t$$
$$= 2^t \cdot R'_t \qquad t \geq 0.$$

Then the total number of allocated HIT segments is therefore

$$Exp(T) = \sum_{t=0}^{\infty} 2^t \cdot R'_t$$

$$= \sum_{t=0}^{\infty} 2^t \cdot \sum_{x=1}^{r'} \sum_{y=r'+1}^{n} P_{t-1}(y) \binom{y}{x} 1/2^y.$$

Then, the expected storage utilization is

$$U = \frac{n}{r \cdot Exp(T)}$$

and the expected height of the DT is

$$H = O(\log_2 Exp(T))$$

Example 1 (Continued)

After inserting the file with twelve keys, the configuration is shown in Figure 3.1. At this time, the statistics of this hash file are:

(a) I (number of internal nodes) = 2
(b) E (number of external nodes) = 3
(c) T (number of allocated segments) = 3
(d) U (utilization) = 12/(6 x 3) = 0.75
(e) H (average height of DT)
   =(3 x 2 + 5 x 2 + 4 x 1)/12 = 1.67
(f) P (number of internal probes)
   $= \sum_{j=1}^{3} \sum_{i=0}^{5} HIT_j[i] = 1.58.$

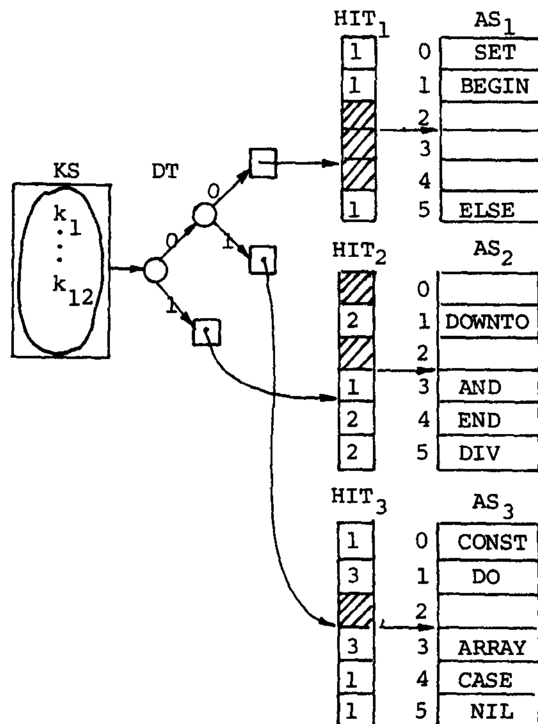


Fig. 3.1 Configuration of the dynamic hash file with twelve keys inserted.

3.2 Simulation Results

Environment and Objectives

First, we produce fifty thousand independent distinct numbers as input data by calling a random number generator. Then we set parameters: the number of rehash functions, s, is 3 or 7, the size of segment, r, is 5 or 10. The hash file considered is created from empty to n=1000. When n is equal to 10,20,30,...,190,200,300,400,500,...,1000, we calculate the following values: the number of internal nodes (I) and external nodes (E) of DT, the number of segments (T) of HIT and AS, the expected height (H) of DT and the number of internal probes (P) in HIT. P and H represent the retrieval cost. The simulation experiment is programmed in Pascal and run on a CDC Cyber 170/720 computer.

Results

(a) Number of nodes in DT

Figure 3.2 shows the expected number of I,E and T where r=5 and s=3,7. The values are the average of 100 tests. From this figure, we observe that:

(1) The expected number of I,E and T are increased in linear proportion smoothy with the number of keys.
(2) In both the cases with s=3 and s=7, the number of segments allocated, T, are fewer than the number of external nodes E. This means some external nodes do not point to a segment of HIT, because in some cases all the keys in a segment were put together when the split occurred.
(3) With the same n, the I,E and T in case of s=7 are smaller than those in case of s=3. This means when using a larger number of rehash functions, the probability of successfully constructing HIT is higher. I.e., the number

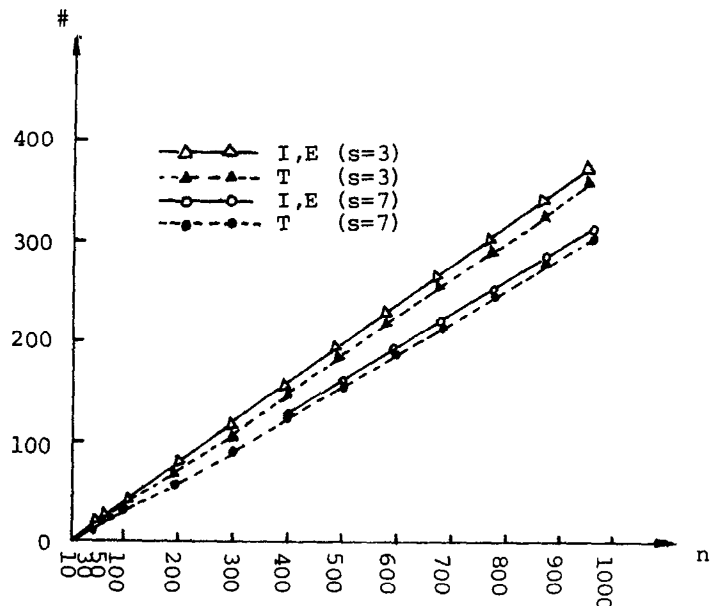


Fig. 3.2 Expected number of I,E and T, where r=5 and s=3,7.

of splits is less.

(b) Utilization

The expected storage utilization for different segments size r, and number s of rehash functions used has been computed and the results are plotted in Figure 3.3. From these results we observe that:

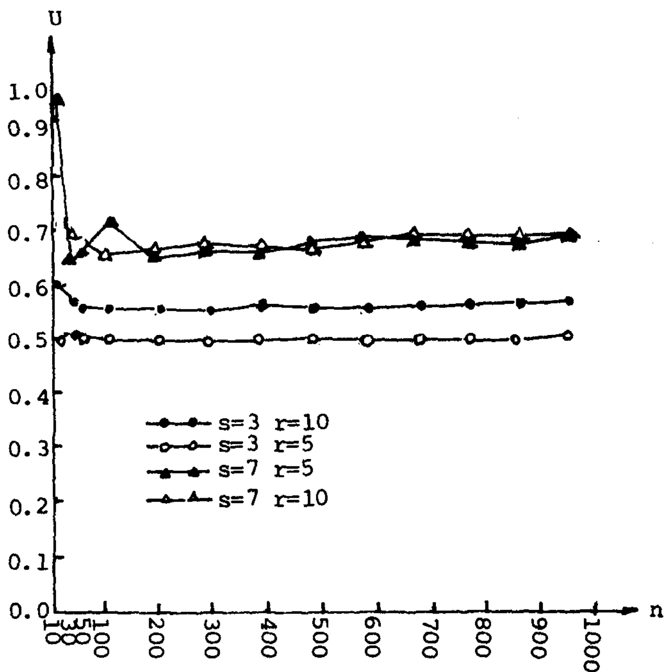


Fig. 3.3 Expected storage utilization (U) for a dynamic hash file.

(1) Storage utilization is almost constant, i.e., the address space is increased steadily and gracefully according to keys actually inserted.
(2) Given a smaller segment, waste can be avoided. With the same s, the utilization is higher when r=5 than that when r=10. However, the number of nodes in DT is increased in case r=5.
(3) With the same r, the storage utilization of s=7 is higher than that of s=3 because we provide more rehash functions. One bit must be added for each entry of segment HIT, but the number of allocated segments is reduced. Therefore, the total size of space is not greatly increased. For example, when r=5, s=7 uses about 158 bits more than s=3 in case of n=300; in case of n=1000, about 588 bits more are used.

(c) Retrieval Cost

The expected retrieval cost are the expected height of DT, and the expected number of internal probes in HIT. For different parameters the results are listed in Figure 3.4. From these curves we observe that:
(1) The height of DT is increased smoothly depending on the number of keys inserted.
(2) The number of internal probes in HIT are steady. This means that we find the hash function value in an almost constant time no matter how many keys are processed.
(3) The differences in average, minimum, and maximum values in the simulation are small. This means the scheme works steadily.

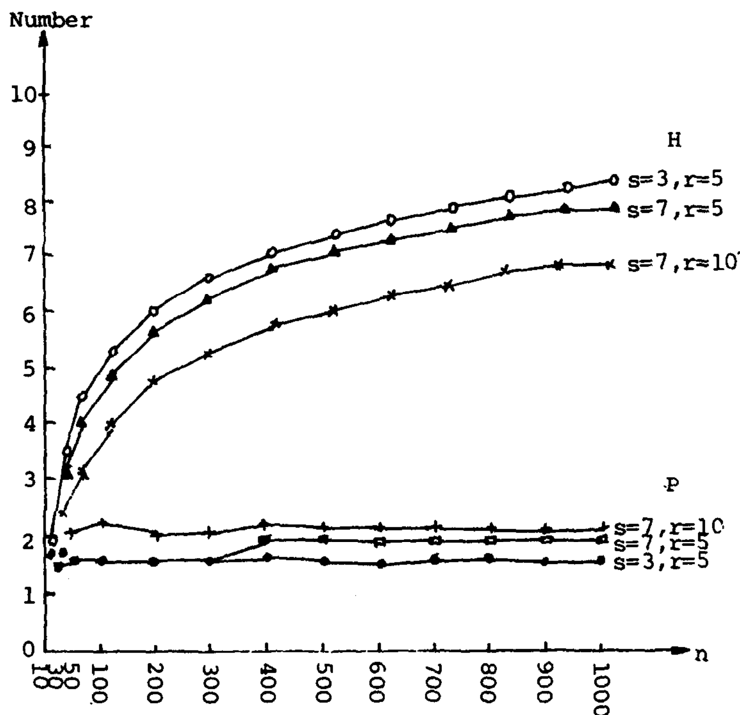


Fig. 3.4 Expected retrieval cost: height of DT(H) and numbers of internal probes (P).

## 4. DISCUSSION AND CONCLUSION

The main feature of this scheme is that it can handle a dynamic key set. We use EHIT to define dynamic perfect hash functions. EHIT contains two parts: distribution tree and HIT. Simulation results show that when the key set is highly dynamic, say n from 1 to 1000, the height of the distribution tree is increased as fast as O(log n), and the number of internal probes and the storage utilization are constants.

In practical, if the auxiliary EHIT can be placed into the main memory, the retrieval operation needs just one disk access to get the associated data in secondary storage. Simulation results show that the size of EHIT is about 484 bytes in case of n=300, s=7, r=5; 1,623 bytes in case of n=1000, s=7, r=5. Therefore putting the EHIT into the main memory is reasonable. Insertion (deletion) operations need one read access and one write access in case no split (merge) occurs. It needs one read access and two write access in case a split occurs and only one read access if a merge occurs in deletion.

## APPENDIX: TO CONSTRUCT HIT

Consider the mapping table as follows:

| | $k_1$ | $k_2$ | $k_3$ |
|---|---|---|---|
| $h_1$ | 2 | 2 | 0 |
| $h_2$ | 3 | 1 | 1 |

Method 1: Process-by-function procedure [6,7]

(1) Select all the singletons from the first row, such as 0 in the following table. An entry in row h and column k is a singleton if there is no other k' such that h(k')=h(k).
(2) Select all the singletons from the second row not in those columns which were selected in the first row, such as 3 and 1 in the following table. We then accomplish and obtain a perfect hash function with HIT=(1,2,0,2).

| | $k_1$ | $k_2$ | $k_3$ |
|---|---|---|---|
| $h_1$ | 2 | 2 | (0) |
| $h_2$ | (3) | (1) | 1 |

Method 2: Process-by-key procedure [28]

Basically, we will process on the keys in the order of $k_1$,$k_2$,etc.

(1) We process $k_1$ first. Since $h_1(k_1)=2$, we circle 2 in the first row of the first column.
(2) Secondly, we process $k_2$. Note that $k_2$ has the same hash value as $k_1$ by applying $h_1$. In this case, $k_1$ and $k_2$ are collided and both need rehashing by $h_2$.
(3) Finally, we process $k_3$. Since zero is a singleton in the first row, we circle it and obtain a perfect hash function with HIT=(1, 2,0,2).

Example A.1 [Failure for both process-by-function and process-by-key methods]

Consider the following mapping table:

| | $k_1$ | $k_2$ | $k_3$ | $k_4$ |
|---|---|---|---|---|
| $h_1$ | 3 | 4 | 2 | 4 |
| $h_2$ | 4 | 3 | 2 | 1 |

The HIT constructed by process-by-function and process-by-key methods are both (0,2,1,1,0). I.e.,

```
3 - 2 -
- - - 1
```

Therefore it does not define a perfect hash function.

In above example, however, we can find two perfect hash functions as defined by HIT=(0,2,1, 2,2) and HIT=(0,2,2,2,2). I.e.,

```
- - 2 -    and    - - - -
4 3 - 1           4 3 2 1
```

These solutions can be obtained by method 3 in the following which is based on branch-and-bound technique.

Method 3: Branch-and-bound procedure [30]

Given a mapping table MT, the procedure "FINDHIT" print all the feasible solutions which define perfect hash functions by using branch-and-bound technique. The procedure is listed in Figure A1. The variables used are defined as follows:

N is the number of the key set.
R is the size of the address space.
S is the number of rehash functions.
I is the index of the value in the solution vector.
MT is the mapping table.
X is the solution vector.
INDEX contains row numbers for the corresponding x-values in the mapping table (i.e., the index of hash functions).
NEXT NEXT(i) points to the next row to be tried in column i.
BACKUP is the number of columns to go back in the problem states.

```
PROCEDURE FINDHIT;
Var MT:array[1..S,1..N] of integer;
    X,INDEX,NEXT:array[1..N] of integer;
    HIT:array[1..R] of integer;
    I,K,BACKUP: integer;
FUNCTION T(I):boolean;//generating the next
              problem states//
  begin T:=false;
    if NEXT[I]≤S then
      begin INDEX[I]:=NEXT[I];X[I]:=MT[INDEX
                 [I],I];
            NEXT[I]:=NEXT[I]+1; T:=true end
  end;
  FUNCTION B(I):boolean;//bounding function//
    begin B:=true; BACKUP:=0;
      for K:=1 to I-1 do
        if X[K]=X[I]
        then begin B:=false; //violate cons-
                    traint (a)//
           if INDEX[K]=INDEX[I] //with same
                    hash function//
           then if INDEX[K]=1 //h1 is used
                    in the first row//
                then BACKUP:=I-K//backtrack
                    k-i columns//
                else BACKUP:=1//backtrack
                    to previous column//
           end
        else if (NEXT[K]>NEXT[I]) AND(MT[INDEX
             [I],K]=X[I])
           then B:=false//violate constraint
                   (b)//
    end;
  Begin//main program starts here//
    while not end-of-file do
     begin
       read(MT); //read mapping table//
       for I:=1 TO N DO NEXT[I]:=1, I:=1;
              //initializing//
       while I>0 do
         begin
           if (T(I) and B(I))
           then if I=N then print (X and HIT)
                //obtain a feasible solution//
                       else I:=I+1 //consi-
                            der the next
                            column//
           else begin //actual backtracking
                     is executed//
              while BACKUP > 0 do
                begin NEXT[I]:=1; I:=I-1;
```

```
                    BACKUP:=BACKUP-1 end;
                  while NEXT[I] S do
                    begin NEXT[I]:=1; I:=I-1
                          end;
              end
          end
       end
    end.
```

Fig. A.1 The Procedure FINDHIT prints all the feasible solutions.

<u>Generating the Next Problem State, T</u>

The boolean function T(i) is used to test whether $(t_1,t_2,\ldots,t_i)$ or corresponding $(x_1,x_2,\ldots,x_i)$ is in the problem state. If it is true, it implies that $x_i$ (where $x_i=h_{t_i}(k_i)$) is assigned from one value of column i in the MT, and $(x_1,x_2,\ldots,x_{i-1})$ have already been chosen. Now the hash value vector is extended to i values, and $x_i$ (to be called the <u>value of current state or current value</u> in short) is considered as a component of a feasible solution.

<u>Bounding Function, B</u>

Bounding function B(i) is false for a path $(x_1,x_2,\ldots,x_i)$ only if the path cannot be extended to reach an answer node, i.e., the bounding function $B_i$ returns a boolean value which is true if the ith x can be selected in the column i of the mapping table MT, and can satisfy the constraints:

(a) $x_i \neq x_j$ for all $i \neq j$, $1 \leq i, j \leq n$.
(b) $x_i \neq d_j$ where $d_j=h_t(h_j)$, $x_i=h_t(k_i)$, $x_j=h_{t'}(k_j)$, if $t < t'$.

Thus the candidates for position i of the solution vector X(1..n) are those values which are generated by T(i) and satisfy B(i). If i=n, we obtain a feasible solution and then print it.

<u>Example A.2</u> [Branch-and-bound method to construct HIT]

Consider the following mapping table:

| | $k_1$ | $k_2$ | $k_3$ | $k_4$ |
|---|---|---|---|---|
| $h_1$ | 1 | 1 | 2 | 1 |
| $h_2$ | 3 | 0 | 2 | 4 |

The size of the solution space is 16. By using the procedure FINDHIT, two feasible solutions are obtained as

```
- - 2 -    and    - - - -
3 0 - 4           3 - 2 4
```

with HIT=(2,0,1,2,2) and HIT=(2,0,2,2,2) respectively. The first solution is the optimal with retrieval cost equal to 1.75.

<u>REFERENCES</u>


1. Anderson, M. R., and Anderson, M. G. Comments on Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets. <u>Comm. ACM 22</u>, 2(Feb. 1979), 104.

2. Chang, C. C., The Study of an Ordered Minimal Perfect Hashing Scheme, to appear in <u>Comm. ACM</u>.

3. Cichelli, R. J., Minimal Perfect Hash Functions Made Simple. <u>Comm. ACM 23</u>, 1(Jan. 1980), 17-19.

4. Coffman, E. G., Jr., and Eve. J. File Structures Using Hashing Functions. <u>Comm. ACM 13</u>, 7(July 1970), 427-432.

5. Cook, C. R., and Oldehoeft, R. R., A Letter Oriented Minimal Perfect Hashing Functions. <u>ACM Trans. on SIGPLAN NOTICES, 17</u>, 9(Sept. 1982), 18-27

6. Du, M. W., Jea, K. F., and Shieh, D. W., The Study of A New Perfect Hash Scheme. <u>Proc. COMPSAC'80</u>, Chicago, Oct. 1980, 341-347.

7. Du, M. W., Hsieh, T. M., Jea, K. F., and Shieh, D. W., The Study of a New Perfect Hash Scheme. <u>IEEE Trans. on Software Engineering, SE-9</u>, 3(May 1983), 305-313.

8. Fagin, R., Nivergelt, J., Pippenger, N., and Strong, H. R., Extendible Hashing - A Fast Access Method for Dynamic Files. <u>ACM Trans. Database Syst</u>. 4, 3(Sept. 1979), 315-344.

9. Jaeschke, G., and Osterburg, G., On Cichelli's Minimal Perfect Hash Functions Method. <u>Comm. ACM 23</u>, 12(Dec. 1980), 728-729.

10. Jaeschke, G., Reciprocal Hashing: A Method for Generating Minimal Perfect Hashing Functions, <u>Comm. ACM 24</u>, 12(Dec. 1981), 829-833.

11. Knott, G. D., Expandable Open Addressing Hash Table Storage and Retrieval. <u>Proc. ACM SIGFIDET Workshop on Data Description, Access, and Control</u>, 1971, 186-206.

12. Knuth, D. E., <u>The Art of Computer Programming, Vol. 3, Sorting and Searching</u>. Addison-Wesley, Reading, Mass., 1973.

13. Larson, P. A., Dynamic Hashing. <u>BIT 18</u>, (1978), 184-201.

14. Larson, P. A., Linear Hashing with Partial Expansions. <u>Proc. 6th Conf. on Very Large Data Bases</u>, Montreal, Oct. 1980, 224-232.

15. Larson, P. A., Performance Analysis of Linear Hashing with Partial Expansions. <u>ACM Trans. on Database Systems</u>, 7, 4(Dec. 1982), 566-587.

16. Litwin, W., Virtual Hashing: A Dynamically Changing Hashing. <u>Proc. 4th Conf. on Very Large Data Bases</u>, West Berlin, Sept. 1978, 517-523.

17. Litwin, W., Linear Hashing: A New Tool for File and Table Addressing. <u>Proc. 6th Conf. on Very Large Data Bases</u>, Montreal, Oct. 1980, 212-223.

18. Litwin, W., Trie Hashing: Res. Rep. MAP-I-014, I.R.I.A. Le Chesnay, France, 1981.

19. Maurer, W. D., An Improved Hash Code for Scatter Storage. Comm. ACM 11, 1(Jan. 1968), 35-37.

20. Maurer, W. D., and Lewis, T. G., Hash Table Method. Computing Surveys 7, 1(Mar. 1975), 5-19.

21. Mendelson, H., Analysis of Extendible Hashing. IEEE Trans. on Software Engineering, SE-8, 6(Nov. 1982), 611-619.

22. Morris, R., Scatter Storage Techniques. Comm. ACM 11, 1(Jan. 1968), 38-44.

23. Ramamohanarao, K., and Lloyd, J. W., Dynamic Hashing Schemes. The Computer Journal, 25, 4(1982), 478-485.

24. Scholl, M., New File Organization Based on Dynamic Hashing. ACM Trans. on Database Systems, 6, 1(March 1981), 194-211.

25. Severance, D. G., Identifier Search Mechanisms: A Survey and Generalized Model. Computing Surveys 6, 3(Sep. 1974), 175-194.

26. Sprugnoli, R., Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets. Comm. ACM 20, 11(Nov. 1977), 841-850.

27. Tamminen, M., Extendible Hashing with Overflow. Information Processing Lett. 15, 5 (Dec. 1982), 227-232.

28. Yang, W. P., Du, M. W., and Tsay, J. C., Single-Pass Perfect Hashing for Data Storage and Retrieval. Proc. 1983 Conf. on Information Sciences and Systems, Baltimore, Maryland, May. 1983, 470-476.

29. Yang, W. P., and Du, M. W., Expandable Single-Pass Perfect Hashing. Proc. of National Computer Symposium, Taiwan, Dec. 1983, 210-217.

30. Yang, W. P., and Du, M. W., A Branch-and-bound Method to Construct Perfect Hash Functions from a Set of Mapping Functions. Research Report of NCTU, Jan. 1984.

31. Yao, A. C., A Note on the Analysis of Extendible Hashing, Information Processing Lett. 11, 2(1980), 84-86.